A Short Introduction to GAUSS*

Ruud H. Koning
rhkoning@xs4all.nl

November 20, 1996

1 Introduction

During the last decade, powerful desktop computers have become available to most researchers. Mainframe computers have been replaced by by personal computers and user friendly numerical programs have replaced old FORTRAN compilers and punch cards. As far as applications are concerned, there is a trend from writing a program for each problem to the use of professional software like statistical programs (SPSS, SAS), spreadsheet programs (Excel, Quatro), and matrix programming languages (MatLab, GAUSS). Using a matrix programming language is attractive compared to programming in a language like C++ or Pascal because the matrix programming language can be optimized for dealing with matrices (and hence it is faster) and because the user does not have to implement the datatype 'matrix of numbers'. In this manual we focus on one particular matrix programming language: GAUSS[1]. GAUSS is not a programming language in the sense that it is not possible to create executable files. A GAUSS program is compiled to pseudo-code and this pseudo-code is interpreted by the GAUSS interpreter.

GAUSS is available for different platforms. It has been developed primarily for MSDOS-based computers, and it has been ported to UNIX during the last few years. At the moment of writing, new versions have been anounced for the windows95/win32s platform as well for Linux.

GAUSS is a very convenient matrix programming language for doing economic and econometric research. It is somewhat less suitable as a data management program, 'specialized' statistical programs like SPSS or SAS are more suited to that task. A major advantage of GAUSS to programming languages is the availability of libraries for specific tasks and the availability of many built-in functions useful to econometricians. There are libraries available for maximum-likelihood estimation, making graphics, numerical optimization, count data analysis, etc. Apart from the commercial libraries, one can also find libraries on the internet. Moreover, most authors of scientific papers are willing to share their code.

There is some GAUSS support on the internet. First of all, there is the GAUSS mailing list (subscribe by emailing to `majordomo@eco.utexas.edu`) which is archived at `gopher://mundo.edo.utexas.edu`. GAUSS source code may be found at the GAUSS software archive at the American University

```
http://www.american.edu/academic.depts/cas/econ/
 software/gauss
```


[1]The version this manual deals with is GAUSS3.2.13. Most of the commands covered work well with previous versions

The distributor of GAUSS (Aptech Systems) provides some support by mail, they can be reached at `info@aptech.com`. Information on libraries and new products can be obtained from their WWW-site at `www.aptech.com`.

In this manual we give a short introduction to GAUSS. Two things should be kept in mind. First of all, this manual is no substitute for the complete GAUSS manual but I hope that the reader is able to use the latter more efficiently after reading this one. Many more command and procedures than the ones discussed in these notes are available. Second, the only way to become proficient in a computer language is by making lots of errors. I recommend making the exercises. The setup of this introduction is as follows. In section 2 we discuss configuring GAUSS. Data types and operators are discussed in section 3 and how to write programs and procedures in section 4. We deal briefly with writing GAUSS libraries in section 5. File handling is treated in section 6. The optional GAUSS module `maxlik` is treated in section 7. It is possible to make publication quality graphics in GAUSS, this topic is dealt with in section 8. Exercises are given in section 9.

2 Configuring GAUSS

GAUSS runs on 386 (or better) computers with a mathematical coprocessor. Memory requirements depend on the type of applications the user has in mind. The program and some libraries use around 7Mb harddisk. Of course, the harddisk requirements increase if the user writes programs and libraries himself, or if he processes large data sets with GAUSS.

GAUSS uses a virtual memory system to create the amount workspace necessary, that is, the user declares the amount of workspace needed and if regular RAM is not sufficient GAUSS 'creates' additional memory by swapping to disk. The virtual memory system must be configured before GAUSS is started. The memory system is configured by calling the batch file `vmi.bat`[2] with three parameters. The first parameter is the amount of workspace required (in Mb's), the second parameter is the directory where the swap file will be located and the third parameter is the amount of RAM memory that can be used by GAUSS (set this parameter to # if GAUSS can use all the amount of RAM available). For example

```
vmi 6 c:\tmp #
```

configures the virtual memory system with 6 Mb workspace, the swap file will be located in the directory `c:\tmp` and all the extended memory available will be used by GAUSS. To avoid excessive disk swapping it is recommended to set the amount of workspace required to the amount of extended memory available on the system minus 2 Mb. In that case, probably all operations can be performed in RAM which is much faster than swap memory. The swap file is deleted automatically if GAUSS is left by pressing `<esc>`. However, the swap file is not deleted if GAUSS is left by rebooting the computer. The user might want to check the swap file directory at times to see whether old swap files are still existent[3]. If swap files from previous GAUSS sessions are remaining they can be deleted without problems. The user might want to check the allocation on his harddisk using the program `chkdsk.exe` or `scandisk.exe` regularly.

GAUSS itself is started by typing `gaussi.exe` at the DOS-command prompt[4]. During startup, the program configures itself according to some variables set in the file `gaussi.cfg` which can be altered by the user[5]. After the program has started it executes the program file `startup` that can be found in the main GAUSS-directory. If the user want some tasks to be done automatically at startup, they can be included in that file. After executing the

[2] This file is in the same directory as the main GAUSS program `gaussi.exe`.

[3] Swap files have names like `jgffwggd`.

[4] Windows95 users can start GAUSS by clicking on the file `gaussi.exe` in Explorer, or by creating a shortcut to this file.

[5] Note that the comments in `gaussi.cfg` use the word 'paths' where most computer users would use the word 'subdirectory'.

command mode		edit mode	
<alt>-z	DOS-shell	<alt>-z	DOS-shell
<alt>-h	help	<alt>-h	help
F7	error log file	<alt>-l	toggle block
F9	last help screen	F9	last help screen
F10	log file	grey +	copy block to scrap
<ESC>	exit GAUSS	grey -	move block to scrap
F1	previous screen	<alt>-w	write block to file
F2	run command	<alt>-p	print block
F3	command start character	<alt>-r	read file at cursor
F4	command stop character	<alt>-g	go to line
<ctrl>-F1	edit last run file	<alt>-s	search
<ctrl>-F3	edit last output file	<alt>-t	search and replace
<ctrl>-<Enter>	carriage return without execute	<alt>-x	exit editor
<ctrl>-t	toggle translator	F1	save and exit
		F2	save and run

Table 1: Some important keys in command- and edit-mode

`startup` file, GAUSS returns with a command start character `>>`. The program is now ready to execute commands entered by the user.

GAUSS can be started in batch mode by `gaussi program.prg`. In this case GAUSS starts and it runs the program `program.prg`. After the program is finished it returns to command mode with the command start character `>>`. If GAUSS is called with the additional parameter `-b` as in `gaussi program.prg -b` it returns to the operating system after the program is finished.

GAUSS can be run in command mode or in edit mode. In command mode, commands are executed immediately after the `<Enter>` key is hit, or after multiple GAUSS commands are ended with the command stop character `<<`. GAUSS commands are separated by a semicolon `;`. In edit mode, commands are typed in a file and they can be stored for later use. That particular file can contain a program, a procedure, etc. Programs stored in a file can be run later. An example of a simple statement in command mode is

```
>>x=rndn(20,2);y=vcx(x);print y;
```

which is to be executed by pressing `<enter>`. This line consists of three GAUSS commands, seperated by semicolons. After pressing `<enter>` the commands are compiled and executed. If the commands do not fit on a single line one can type further on a new line after pressing `<ctrl><enter>`. All commands will be executed by either pressing `<enter>` or by ending the command with the command stop character `<F4>` (`<<`) followed by the run command `<F2>`. The example can also be run in edit-mode. The edit-mode is started by typing `edit example1.prg` at the GAUSS command prompt. This command starts the GAUSS editor and opens (or creates) the file `example1.prg`[6]. The three commands of the example can now be entered as

```
x=rndn(20,2);
y=vcx(x);
print y;
```

Pressing `<return>` in the edit mode does not execute the statement at that particular line, it starts a new line instead. After the program has been entered, one can leave edit-mode by pressing `<alt><x>`. A bar appears with six options available: Write, Quit, eXecute, Debug, Run options, and Compile options. Pressing `<x>` at this moment compiles and executes the program. Pressing `<d>` compiles the program and runs it in debug mode so that

[6] Usually one does not want GAUSS programs to be stored in the directory where `gaussi.exe` is located. One can change to another directory by giving the command `>> dos cd prog` in command mode, which changes the active directory to `c:/gauss/prog`. If one wants to start in this directory everytime GAUSS is started, one must add this line to the file `startup`, see above.

the user can monitor execution of the program. After executing the program, GAUSS returns to the command prompt >>. In another GAUSS-session one might want to modify the program `example1.prg`. This is done simply by starting the editor again: `>>edit example1.prg`.

The most important keystrokes in command- and edit-mode are given in tabel 1 but of course this list is not complete.

Online help is available within GAUSS by pressing `<alt><h>`. After the help engine has been started, one can get more information on special key strokes in command mode by `h` followed by `@cmkeys` and more information on special keys in edit mode is obtained by `h` followed by `@ekeys`. Help is on functions of libraries (see section 5) is available if the library is active.

GAUSS is left by pressing `<esc>` in command-mode.

3 Data Types and Operators

GAUSS knows only two data types and it is not possible to have new data types defined by the user. The two data types are matrices and strings. Matrices are two-dimensional arrays. The elements of a matrix must be either numbers (numbers are stored in double precision, so that the number of significant digits is 15 or 16, numbers must be in the range $4.19E-307 <\mid x \mid < 1.67E308$) or characters. In the latter case the elements must be strings of up to eight characters so that the amount of memory required for each element does not exceed 8 bytes. A matrix may consist of elements of both types, for example a data matrix may have characters with variable names in the first row and the remaining rows can contain the data. In the virtual memory version of GAUSS there is no limit on the size of the matrix apart from the amount of workspace available. The other data type is the string data type. Again, there is no limit to the length of the string, apart from the workspace available.

GAUSS does not type check the variables, so it is possible to run the following program without encountering any errors:

```
s="this is a string";
s=s+5;
```

A matrix can be initialized in five ways. First, it can be initialized using the `let`-statement as in `let x={ 1 2, 3 4}` or in `let x[2,3]= 1 2 3 4 5 6` where the matrices are filled row-wise. Second, a matrix can be initialized by concatenation of existing matrices, for example

```
let a={1 2, 3 4};
let b={5 6};
let c={10, 11, 12};
x=a|b;
x=x~c;
```

Here, `a` is a 2×2-matrix, `b` is a 2-row vector and `c` is a column vector of length 3. Data are read row-wise and consecutive rows are seperated by a colon. The matrix `x` is initialized by vertical concatenation using the `|`-operator and then the vector `c` is added by horizontal concatenation with the `~`-operator. It is possible to initialize an empty matrix by `let x={}` and to concatenate matrices to this matrix `x`: `x=x~a`. After this initialization, `x` equals `a`. The third way of initializing a matrix is by using special matrix functions like `x=ones(n,k)`, `x=zeros(n,k)`, etc. The fourth way of initializing a matrix is by keyboard input: `x=con(2,3)`. After this command, the user is asked to enter the elements of the matrix interactively. The last way to initialize a matrix is by reading the matrix from disk if that matrix has been saved from a previous GAUSS session. A matrix saved in a GAUSS session can be retrieved by `load x=soccer`. The contents of the file `soccer.fmt` are read into `x`. ASCII data can be loaded analogously by `load x[]=soccer.asc`. This

method assumes that the file `soccer.asc` contain ASCII data only. The file is read rowwise into a column vector `x`. If necessary, this column vector can be reshaped in a matrix with the command `x=reshape(x,100,5)`. For more details on file input/output we refer to section 6. A matrix can be printed on screen or to another output device (like a file or printer) by typing its name: `x` followed by `<enter>` gives

```
x;

       1.0000000        2.0000000        10.000000
       3.0000000        4.0000000        11.000000
       5.0000000        6.0000000        12.000000
```

in the example above. Code that is better readable is `print x;` which yields the same output.

A matrix with characters can be initialized by explicit initialization as in `a={"x1","x2"}` or, quivalently, `a="x1"|"x2"`, by concatenation, or by reading from disk. In order to print the vector `a`, it must be preceded by a `$`-sign as in `$a`.

Strings can be initialized analogously to matrix initialization. A string may be initialized by explicit assignment, as in `s="this is a string"`. Existing strings can be concatenated to a new string:

```
string1="this is a string";
string2="and this a second string";
string3=string1$+string2;
```

A string can be initialized by keyboard input with the command `string=cons`. Note that it is not necessary to specify the (single) dimension of the string. The appropriate amount of memory to store the string will be allocated during the input from the keyboard. Strings saved to disk in a previous GAUSS session are retrieved by `loads s=string`. The contents of the file `string.fst` are read into `s`. A string can be printed by typing its name (without a `$`-sign preceding it): `string3` followed by `<enter>` will print `this is a string and this is a second string`.

Certain character combinations are not allowed in strings. In general, the backslash `\` indicates an escape character. Sometimes a backslash is needed in a string (for example to indicate a path). In that case, one should use a double backslash as for example in `path=c:\\gauss`. Some other special characters characters are `\b` (backspace), `\e` (escape), `\f` (formfeed), `\g` (beep) and `\t` (tab). `\123` generates a character whose ascii-value is '123'.

One must use the caret (`^`, also known as the substitution operator) in order to get a filename from a string variable. Consider the following example:

```
data="dataset";
load x1=data;
load x2=^data;
```

The matrix `x1` will contain the data stored in a file with the name `data` and the matrix `x2` will contain the data stored in the file `dataset`. In the first case, `data` is interpreted as a literal and in the second case it is substituted by the contents of the string.

Strings, character matrices and numbers can be converted into each other. An example may be useful. Let `c` be a vector with character elements. An element of `c` can be transformed into a string by concatenating it with `""`. Similarly, a string can be transformed into a character element by preceding it with 0. See the following example:

```
c="x1"|"x2";            /* c is character vector */
s=""$+c[1];             /* s is string variable */
z="string";             /* z is string */
b=0$+z;                 /* b is character vector with one
                           element */
```

It is mandatory that `""` and `0` come first in the concatenation. A numerical value can be transformed in a character value using the function `ftocv` and into a string using `ftos`. Conversely, a string can be transformed into a numerical variable by the function `stof`. The following lines create a character vector with character elements `"var1"` to `"var10"`:

```
n=seqa(1,1,10);      /* n is a 10-vector with 1,2,...,10 */
var_n=0$+"var"$+ftocv(n,1,0);
```

The second argument of the function `ftocv` is the minimum field width (1 in this case, as the shortest number is represented by 1 character) and the last element indicates the number of decimal places.

GAUSS distinguishes between 'regular' and element-by-element operators that are defined on both datatypes. In due course, it will be shown that the latter ones can be very convenient. The more important regular operators defined on matrices are `+` (addition), `-` (substraction), `*` (matrix multiplication), `/` (division, `x=b/A` is the solution to `Ax=b`), `%` (modulo division) and `!` (factorial). All these operators are defined on matrices of appropriate dimensions only. Element-by-element operators are performed elementwise. Examples are `.*` (element-by-element multiplication), `./` (element-by-element division), `.^` (element-by-element exponentation, the same as `^`) and `.*.` (Kronecker product). Element-by-element operators are obtained by preceding the regular operator with a dot `.`. Other important matrix operators are `'` (transpose), `~` (horizontal concatenation) and `|` (vertical concatenation). Two important operators defined on strings and character matrices are `$+` (string concatenation) and `^` (string variable subsitution).

Various relational operators are defined in GAUSS. Most operators can appear in two forms other than their 'regular' form: the element-by-element form and the `$` form for comparisons between character data and strings. The element-by-element form is obtaind from the regular form by preceding that form with a dot `.`. The relational operators available are `==` (is equal to), `<` (is less than), `<=` (is less than or equal to), `>` (is greater than), `>=` (is greater than or equal to) and `/=` (is unequal to). These expressions will evaluate to 1 (true) or 0 (false) if they are used in their regular forms. If the relational operator is used in its element-by-element form it evaluates to a matrix of 0's and 1's, depending on whether the condition holds for that pair of elements. If `x` and `y` are matrices of the same dimensions with floating point numbers, `x==y` will evaluate to 1 or 0 (depending on whether *all* elements of `x` and `y` are equal or not) and `x.==y` will evaluate to a matrix (of the same dimensions as `x` and `y`) with 1's and 0's.

Finally, the following logical operators are implemented: `AND`, `OR`, `NOT`, `XOR` (exclusive `OR`) and `EQV`. Again, these operators can be used element-by-element by preceding them with a dot `.`.

An example may illustrate the use of an elementwise operator. The following program simulates data for a probit model:

```
nobs=100;                /* number of observations */
true_beta=0|1|-1;        /* vector of parameters */
x=ones(nobs,1)~2*rndn(nobs,2);     /* x matrix with intercept and
                                      regressors */
y=(x*true_beta+rndn(nobs,1)).>=0;  /* vector with 0's and 1's */
```

In the last line, every element of `y` is set to `0` or `1` depending on whether the corresponding element of the vector `x*true_beta+rndn(nobs,1)` is smaller than 0 or not. This code is much faster and certainly more transparent than 'usual' code with some kind of loop. Elementwise operators should be used as much as possible as they are much faster than code corresponding code performing the same task for each element individually.

Sometimes one needs a submatrix of a given matrix. The desired rows and columns can be selected by indicating them between square brackets: `x[.,1]` selects the first column of a matrix `x`, `x[1:4,.]` selects the first four rows of `x`. The upper left 4×4 block of `x` is selected by `x[1:4,1:4]`. It is also possible to extract a submatrix using a vector with indices as in

```
c=1|4|6;
a=x[c,.];
```

which selects rows 1, 4, and 6 into a new matrix `a`. Of course, the maximal element of `c` should not exceed the number of rows of `x` and all elements of `c` should be positive integer numbers.

4 GAUSS Programs and Procedures

GAUSS programs can loosely be defined as files with valid GAUSS commands. A program file may consist of both the actual code of the program and additional procedures specific to that program. As a first command of a GAUSS program, the user can start to give `new;` which clears the workspace. All matrices and procedures from previous programs are deleted from memory. A program can be ended with the `end;` statement which closes all open files and terminates the program. The `pause(10);` statement halts execution of the program for 10 seconds and the `system;` statement exits GAUSS.

An important element in any program is the flow control. Various keywords are available in GAUSS to determine whether a piece of code should be repeated some times or should be executed at all. A GAUSS-loop is started using the `do-while` statement and ended by the `endo` statement. Within the loop, one can jump to the top of the loop with the `continue`-statement and one can break out of the loop with the `break`-statement. In that case the program proceeds with the first command following `endo`. Consider the following example:

```
i=1;
do while (i<=100);
 i=i+1;
 print "i=" i;
endo;
```

Note that a scalar expression `i<=100` determines whether the loop should be executed again or not. Alternatively, the loop can be controlled using the `do-until`-statement.

It is not advisable to perform matrix operations using a `do-while` loop if they can be performed using elementwise operators instead. The following program generates 10000 random numbers uniformly distributed between 0 and 10 and classifies them into the intervals $[0, 3)$, $[3, 7)$, and $[7, 10]$. (`hsec` is a standard GAUSS function that returns time elapsed since midnight in hundredths of a second.)

```
r=10000;
v=10*rndu(r,1);

et1=hsec;
v1=zeros(r,1);
i=1;
do while (i<=r);
 if v[i]<3;
  v1[i]=1;
 elseif v[i]>7;
  v1[i]=3;
 else;
  v1[i]=2;
 endif;
 i=i+1;
endo;
et1=(hsec-et1)/100;

et2=hsec;
v2=(v.<3) + 2*(v.>=3).*(v.<=7) + 3*(v.>7);
```

```
et2=(hsec-et2)/100;

print "loop" et1;
"vectorized" et2;
"ratio" (et2/et1);
```

Classification is much faster if done using the elementwise operators.

Code is executed conditionally using the `if-endif`-statements. Within an `if-endif`-branch one can use `elseif` and `else`-statements for further conditioning. Both the `if` and `elseif`-statements must be followd by a scalar expression which determines whether the code should be executed or not. Each `if`-statement must be ended with an `endif`-statement. Consider the following example:

```
i=1;
do while (i<=20);
 if (i%2==0);
  print "i is even" i;
 elseif (i%3==0);
  print "i is odd and divisible by 3" i;
 else;
  print "i is odd and not divisable by 3" i;
 endif;
 i=i+1;
endo;
```

Procedures are the building blocks of GAUSS. Many useful procedures are provided with the installation and other handy procedures are shipped with the GAUSS modules or with commercial extensions to GAUSS. GAUSS derives its flexibility from the possibilities for the user to write his own procedures. These procedures can be very simple or complex, even though it is recommendable to break complex procedures in a few simpler ones. In the remainder of this section we will discuss writing a procedure and some important standard procedures that belong to the standard installation of GAUSS.

A general procedure is created with the following steps. First, the source code must be written in a file[7]. That file must have the same name as the procedure[8], and have extension `.g`. For example, a procedure `boxcox` must be in the source file `boxcox.g`. This requirement implies that a procedure name can have up to eight characters. The file with the source code must be placed in a subdirectory that is listed in the path for program files (the variable `src_path` in `gaussi.cfg`, see section 2).

The actual code for the procedure consists of the following five parts:

1. procedure declaration,
2. declaration of the local variables,
3. actual code of the procedure,
4. returning values,
5. end of the procedure.

It is good practice to document the most important features of the procedure in the first couple of lines, between the comment terminators `/*` and `*/`. If the user needs help on the procedure he can use the GAUSS help system as if the procedure were an intrinsic command or a vendor provided procedure. The help system puts the source code of the procedure in a window, hence documentation should be in the top of the source file.

[7] If the procedure is specific to one program (for example, a procedure that calculates a specific likelihood function to be optimized) the code of the procedure may also be placed in the file that contains the code of the GAUSS program. In this case, that procedure can not be called from other programs.

[8] If the procedure is part of a library, this is not necessary. Creation of libraries is discussed in section 5.

Every procedure starts with a declaration, as for example `proc (1) = boxcox (x, l);`. The number of returns is put between parentheses, if only one object is returned this can be left out as in `proc boxcox(x,l);`. The parameters of the procedure are passed after the procedure name. In this case, there are two parameters.

The second element of the procedure is the declaration of the local variables. All variables in GAUSS are global ones, unless they are preceded by the keyword `local` when they are declared. All global variables are accessible from within the procedure and are not declared in one way or another. Local variables are declared as in `local z;`. After this command, `z` can be initialized, it is not initialized by its declaration. A local variable may have the same name as a global variable, within the procedure where it has been declared as the local variable temporarily 'overrides' the global variable of the same name.

The middle part is generally the most interesting part. In this part the actual calculations are performed. In this section other procedures may be called.

The fourth element of the procedure is returning the result of the calculations. The number of elements returned must coincide with the number of returns given in the declaration. An example of the return statement is `retp( z );`. If the procedure has no returns, this part can be skipped, if more than one element is returned, the elements are separated by comma's as in `retp( z, x );`. Finally, a procedure is terminated with the `endp;` command.

An example of a procedure that calculates the Box-Cox transform is

```
/* help comments here
*/

proc (1)=boxcox1(x,l);
 local z;
 if (rows(l)/=1 or cols(l)/=1);
  errorlog "boxcox1.g: l must be a scalar";
  retp( -1 );
 endif;
 if (l==0);
  z=ln(x);
 else;
  z=(x.^l-1)/l;
 endif;
 retp( z );
endp;
```

This procedure could be extended with some help comments on top. A slightly quicker version of this procedure is

```
/* boxcox2.g: procedure to calculate the Box-Cox transform
input:  x: n x k matrix with positive elements,
        l: scalar,
output: y: n x k matrix with Box Cox tranformation of each
           element of x
*/

proc (1)=boxcox2(x,l);
 if (rows(l)/=1 or cols(l)/=1);
  errorlog "boxcox2.g: l must be a scalar";
  retp( -1 );
 endif;
 if (x>0);
  if (l==0);
   retp( ln(x) );
  else;
   retp( (x.^l-1)/l );
```

```
  endif;
 else;
  errorlog "boxcox2.g: x must be positive";
  retp( -1 );
 endif;
endp;
```

This version uses less workspace because no local variables are declared. Moreover, note that it is possible to exit from the procedure with a `retp`-statement from anywhere within the procedure.

One should note that GAUSS is not a very 'safe' language in the sense that it hardly performs any type checking at either compilation or run time. Hence, the user should do this when he writes the procedure. Sometimes a procedure uses global variables (for example the `ols`- and `dstat`-procedures to be discussed below). It is in general a better idea to pass global variables as parameters to the procedure. This makes the procedure better useable in another context: global variables tend to be there when you need them, but also when you don't expect them. The following program is valid GAUSS code; it shows the dangers of accessing global variables from within a procedure.

```
new;
a=3;
change_a;
print a;

proc (0)=change_a();
 a=5;
endp;
```

The printed result is `5`.

Procedures can be called in different ways:

```
x=boxcox(y,0.5);
boxcox(y,0.5);
{va,ve}=eigrs2(h);
call eigrs2(h);
```

In the first case, the result of the procedure is stored in the matrix `x`. This matrix `x` is created and initialized automatically, if necessary. In the second case, the output of the procedure `boxcox` is copied to screen. The output of the procedure `eigrs2` consists of two elements. The first matrix in the `retp`-statement of `eigrs2` is stored in `va` and the second matrix in that statement in `ve`. In the fourth case, the procedure is executed and all output is discarded. This way of calling a procedure may be useful if the return indicates succesful completion of the procedure only (as is the case with, for example, the procedure `xy`).

A special kind of argument of a procedure is a pointer to another procedure. This can be useful when a procedure is an argument, for example when one writes a general kernel estimation procedure. An example where a procedure is passed as a pointer to another procedure is:

```
/* program to illustrate procedures as arguments to other
procedures
*/

new;
library pgraph;
nobs=10000;
ngrid=500;
y=sortc(rndn(nobs,1),1);
z=seqa(-3,6/ngrid,ngrid);
```

```
f1=kernelestimate(z,y,0.1,&biweight);
f2=kernelestimate(z,y,0.1,&gaussian);
ftrue=pdfn(z);
call xy(z,f1~f2~ftrue);


proc kernelestimate(z,x,h,&kf);
 local arg,i,f,kf:proc;
 i=1;
 f=zeros(rows(z),1);
 do while (i<=rows(z));
  arg=(z[i]-x)/h;
  f[i]=meanc(kf(arg))/h;
  i=i+1;
 endo;
 retp( f );
endp;

proc biweight(x);
 retp( (15/16*(1-x^2)^2).*(abs(x).<1) );
endp;

proc gaussian(x);
 retp( pdfn(x) );
endp;
```

In this example, pointers to the `biweight`- and `gaussian`-procedures are passed as parameters to the procedure `kernelestimate`. Pointers to these procedures are obtained by preceding the name of the procedure with an ampersand (`&`), a convention well-known to C- and C++-programmers. Note that the symbol for the procedure to be passed in the argumentlist of the procedure `kernelestimate` (`kf` in this case) must be declared as a local procedure in the local declaration list (`local kf:proc`). It is the responsibility of the user that `kf` is called correctly.

GAUSS is shipped with many standard procedures. Two important procedures are the procedure to perform linear regression and the procedure to calculate descriptive statistics. Both procedures use both local and global variables. First we discuss the regression procedure. The procedure estimates parameters in the linear model

$$y_i = \beta' x_i + \varepsilon_i, \qquad i = 1, \ldots, N. \tag{1}$$

In equation (1), the k-vector x_i is the vector with regressors that may or may not include a constant term. The disturbances ε_i are assumed to be independently distributed with zero mean and (constant) variance σ^2. The parameters of the model (β and σ^2) can be estimated by

```
{vnam,m,b,stb,vc,stderr,sigma,cx,rsq,resid,dwstat}
 =ols(dataset,depvar,indvar);
```

The parameters of this procedure are `dataset` (a string variable containing the name of the dataset), `depvar` (a character vector with one element or a scalar pointing to the row in the dataset with the dependent variable) and `indvar` (a character vector with the names of the indepenent variables or a vector with the row indices of the independent variables). If `dataset` is a null string, the actual vector of the dependent variable and the matrix with independent variables are assumed to be passed to the procedure as `depvar` and `indvar`. The way the OLS estimates are calculated is partly determined by a few global variables. The first is `__con` (`con` is preceded by *two* underscores) if this variable is set to 0 the regression is estimated without an intercept. The default value is 1 so that an intercept is included. A second global variable is `__miss`, this one determines how missing values are

treated. The default value is 0 so that the procedure assumes all observations are valid. A third important global variable is _olsres (olsres is preceded by one underscore). If this variable is set to 1, the Durbin-Watson test statistic for autocorrelation is calculated and the OLS-residuals are determined. The default value is 0. Finally, the variable __altnam may be set to a character vector with the names of the variables, with the name of the dependent variable as the last element. Detailed information on this procedure can be obtained from the online help by <alt>-h followed by h and ols.

Another useful standard procedure is dstat. This procedure calculates the mean, standard deviation, minimum, maximum and number of valid cases of a dataset or a datamatrix. Its syntax is

```
{vnam,mean,var,std,min,max,valid,missing}
 =dstat(dataset,vars);
```

Again, dataset is a string variable with the name of the dataset to be analysed and vars may be either a character vector or an index vector. If vars is 0, descriptive statistics of all variables in the dataset are listed. If dataset is 0, vars is assumed to be a datamatrix that is analyzed. Treatment of missing values is determined by the global variable __miss. This variable has default value 0 so no checking for missing values is performed.

An example where both the ols- and dstat-procedure are used is

```
nobs=100;
x=ones(nobs,1)~3*rndn(nobs,3);
beta=0|1|-1|0.5;
sigma=1;
y=x*beta+sigma*rndn(nobs,1);

_olsres=1;
__altnam="CONSTANT"|"X1"|"X2"|"X3"|"DEPVAR";
call dstat(0,x~y);
call ols(0,y,x);
```

5 Libraries

GAUSS is a modular language, it can be extended by libraries that add new procedures to the language. These libraries are distributed both as commercial add-ons to GAUSS as well as free software. In this section we will describe how add-on libraries can be used as well as how one can write his own library.

Before we discuss writing libraries, we need to discuss the way GAUSS searches for unknown references (like a matrix or a procedure). If the compiler encounters an unknown object during compilation, the state of the autoloader determines how the program searches to resolve the unknown reference if the reference cannot be resolved in the file under compilation. First, the autoloader searches in the current directory (which is listed in the lower-right corner of the screen) and then along the searchpath given by src_path in the file gaussi.cfg (see also section 2). The exact way how is searched for unknown references is determined by the state of the autoloader and the autodelete state. The autoloader/autodelete state can be toggled in command mode by pressing <ctrl>-a.

Suppose GAUSS encounters this line of code:

```
{b,s2}=ols_estimate(y,x);
```

Since ols_estimate is not an intrinsic GAUSS function, code for this function needs to be located and compiled. If the autoloader is turned off, then the procedure must have been declared before with the command

```
external proc ols_estimate;
```

In that case forward references (ie, references to objects not already defined) are not allowed. If both the autoloader is on and the autodelete-state is on GAUSS searches for the unknown object along the following paths. First, GAUSS tries to find it in the user library, then in user-specified libraries, then in the GAUSS library and finally it searches for files with a `.g` extension in the current directory and along the path listed in `src_path`. In case the autoloader is on but the autodelete-state is off, GAUSS does not search for files with a `.g` extension. Moreover, forward references to objects not listed in the libraries are not allowed. Compilation time is longest when both the autoloader and the autodelete-state are on, but that case is most convenient to the user. In the remainder of this section we assume that both the autoloader and the autodelete-state are on.

A GAUSS library is best thought of as an index file where the program can find the exact location of references. Libraries are activated by a command like `library maxlik, bstat;`. After activation, procedures and matrices defined in these libraries become available in a program. Two libraries are activated by default when GAUSS is started: the `gauss`- and `user`-libraries (unless this feature is turned off using `<ctrl><l>`). The `gauss`-library is a library with native gauss procedures like `dstat`, `bstat` and many others. The `user`-library is empty at the moment GAUSS is installed on ones system, but the user can add his own procedures to this library (see below).

GAUSS libraries are stored as ACII-files with extension `.lcg` in the directory specified by the variable `lib_path` in `gaussi.cfg`. An example of such a library file is `course.lcg`:

```
c:\gauss32\prog\ticourse\testlib.src
    testlibset                      : proc
    ols_estimation                  : proc

c:\gauss32\prog\ticourse\testlib.dec
    _df_correction                  : matrix
```

This library consists of three objects: one matrix and two procedures. If the `course`-library has been activated by `library course` and the compiler encounters a reference to either one of these objects, GAUSS 'knows' where to find the code for that particular object and the file containing that object will be compiled. The list of active libraries can be found by giving the command `library` without any library names. Additional code may be added to the library using the `lib` command. Objects in the file `testlib2.src` may be added by `lib course testlib2.src`. One final remark concerning this example is in order. Global variables in this library are declared in the file `testlib.dec` and are reset to their default values in the procedure `testlibset`. The code in `testlib.dec` is

```
declare _df_correction!=1;
```

and the code in the source file in this library is

```
#include testlib.dec

proc (0)=testlibset;
 _df_correction=1;
endp;

proc ols_estimation(y,x);
 local n,k,sxx,sxy,b,e,s2;
 n=rows(x);
 k=cols(x);
 sxx=x'x/n;
 sxy=x'y/n;
 b=inv(sxx)*sxy;
 e=y-x*b;
```

```
 if (_df_correction==1);
  s2=e'e/(n-k);
 else;
  s2=e'e/n;
 endif;
 retp( b, s2 );
endp;
```

The global variable `_df_correction` is used in the procedure `ols_estimation` and this is the precise reason why this variable is declared in `testlib.dec`. If that variable would not be declared, compilation of `testlib.src` would give an 'illegal redefinition' error or an 'undefined symbol' error.

6 File Input/Output

One of the most troublesome problems with any computer language is corresponding with other programs. GAUSS can read ascii-datafiles and it can read and write datafiles in a binary format that is not recognized by other programs. It is not possible to import binary data files of well-known programs as SPSS or LOTUS.

An ascii-file with numeric data is read into a vector `x` with the `load`-command: `load x[]=file.asc`. The data are read row-wise from `file.asc` and must be seperated by a white space (ie, a space, tab or return). The data in `file.asc` must be data suitable for storage in the GAUSS data type matrix: they must be numerical or a sequence of characters starting with the character `'a'...'z'` or their uppercase equivalents. GAUSS stops processing the input stream as soon as a nonnumerical entry with a non-valid first character is found (for example &) even though valid data may follow.

Compiled procedures, strings, and matrices can all be saved in a binary format. Most often, one wants to save a matrix or a string (array) so that these can be used later for further analysis. The general format of the `save`-command is `save name=symbol;` where `name` is a literal or a referenced string and `symbol` is a symbol (a matrix or a string, for example). Examples are (`x` is a matrix and `s` is a string):

```
save x;
save names=s;
save c:\tmp\xx=x;
save path=c:\tmp x;
```

In the first case, `x` is saved into the binary file `x.fmt` in the current directory, in the second case, `s` is saved into the binary file `names.fst` in the current directory, in the third case, `x` is saved into the file `xx.fmt` in the directory `c:\tmp` and in the final case `x` is saved into the file `x.fmt` in the directory `c:\tmp` and all further objects saved with the `save`-command will be placed in this directory, unless an explicit filename is given as in the third example. The extension `.fmt` indicates a matrix written to disk and the extension `.fst` indicates a string written to disk.

A matrix file created using the `save`-command can be loaded into the workspace using the `load`-command, for example by `load x` (`x.fmt` is loaded from the current directory into `x`) or `load x=c:\tmp\xx` (`c:\tmp\xx.fmt` is loaded into `x`). A binary file containing a string can be loaded with the `loads`-command, which works analogously to the `load`-command. Note that the `load`-command can be used to read ascii-data into a GAUSS matrix. It is not possible to read ascii-data using the `loads`-statement.

A GAUSS dataset may either be created using a conversion utility (`atog386`, see below) or a dataset may be created in a GAUSS session. A dataset can be created in two different manners: from within GAUSS or by using a conversion utility that converts a dataset in ASCII-format into GAUSS format. First, one can use the `saved(x,dataset,vnames)` where `x` is the matrix to be saved in the datafile, `dataset` is a string variable containing

the name of the dataset and `vnames` is a vector with the names of the columns of `x`. Upon succesful completion `saved` returns the value 1. It is customary to assign variable names in capitals for numerical variables and in lowercase for character variables. If no vector with variable names is passed (*i.e.*, `vnames` is set to `0`), GAUSS creates a vector with variable names automatically, with names `X1`, `X2`, etc.

```
/* illustration of writing and reading datasets */

x=3*rndn(1000,5);
y=rndn(1000,1);
c=(y.<0).*"ltzero"+(y.>=0).*"gtzero";
data=x~c;
/* uppercase: numerical variables
   lowercase: character variables */
vnames="X1"|"X2"|"X3"|"sign";
file_name="testdata";
saved(data,file_name,vnames);

z=loadd(file_name);
```

Data can be read into memory using the `loadd`-command as in the example above. This, however, works for small datasets only. The variable names of the columns in the dataset can be retrieved by the command `vnames=getnames(dataset)` where `dataset` is a string variable with the name of the dataset. Alternatively, data can be read row-wise from a dataset, as in the example below.

```
/* illustration of reading datasets row-wise */

file_name="testdata";
vnames=getname(file_name);
i=1;
data={};
open fh=^file_name;
number_rows=rowsf(fh);
do while i<=number_rows;
 data_row=readr(fh,1);
 data=data|data_row;
 i=i+1;
endo;
close(fh);
```

First, one assigns a file handle to the file that is going to be read. Then the dataset is accessed using the `readr(fh,1)`-command. The first argument is the file handle, the second argument is the number of rows that is going to be read. Finally, the dataset is closed. All open files in a GAUSS session may be closed with `closeall`.

A special utility `atog386` is provided for conversion of an ascii-dataset to a GAUSS-dataset. This program can be run both interactively and in batch mode. In the latter case, commands are processed from a batch file and the conversion is performed by `atog386 convert.cmd`. Conversion commands are discussed in Appendix A. After conversion, the dataset can be accessed using the commands given above.

Even though GAUSS may not be the most efficient package for data handling and coding, it has a special facility to do so. Of course it is possible to write a program that reads a dataset, changes or adds some columns and saves the result. However, these tasks are performed more easily using the `dataloop` command. A program using the `dataloop` command is translated to a GAUSS procedure first, and then the resulting code is compiled and executed. For the first step to work, it is necessary to have the translator turned on. This is accomplished by `<ctrl>-t` in command mode. An example of a data loop program is

```
/* program that selects matches of Hiddink */

infile="bondscoa";
outfile="bonds2";
dataloop ^infile ^outfile;
 select d12==1;
 make expvoor=exp(voor);
 drop d1 d2 d3 d4 d5 d6 d7 d8 d9 d10 d11;
endata;
```

The most important statements inside a data loop are: `code` (creates a new variable with coded values), `delete` (specifies which rows can be deleted), `drop` (specifies which variables are not written to the output file), `extern` (declares a symbol to be a matrix or string in memory) `keep` (specifies which variables are kept in the new data file), `lag` (lags a variable one or more periods), `make` (creates a new variable), `recode` (recodes an existing variable in the dataset) and `select` (selects cases). Help on data loop commands is obtained by requesting help on `@dataloop` within the help engine.

7 Maximum Likelihood Estimation

A very convenient GAUSS libray is the maximum likelihood library[9]. This library contains a procedure to maximize a loglikelihood function and a procedure to determine whether a user has programmed a derivative of the loglikelihood correctly. In fact, the library consists of a mere five procedures: `maxlik` for optimization of a likelihood function, `maxset` to set the global variables at their default values, `maxprt` to print the output of the `maxlik` procedure, `maxclr` to set the global variables to their default values if nested copies of `maxlik` are called and `maxgrd` to check whether a gradient procedure yield the same results as some numerical gradient procedures. Note that this library is not distributed with the GAUSS-program itself: it must be bought seperately[10]. Computer language code for global optimization of functions is widely available nowadays (see for instance Press, Teukolsky, Vetterling, and Flannery (1992)), but most of this code works only well with very well behaved functions. The optimization routines in the maxlik-library are based on Dennis and Schnabel (1983). The maxlik-library has many options that can be set by the user, here we will only deal with the most important ones (as judged by the author) and even then not all options will be discussed. The reader is advised to read the Maximum Likelihood Manual for more information. Of course, on-line help is available as soon as the maxlik-library has been activated.

We will discuss the use of the maxlik-library by means of a simple example. Consider the normal linear regression model with the data generated according to

$$y_i = \beta' x_i + \varepsilon_i, \qquad \varepsilon_i \sim \mathcal{N}(0, \sigma^2). \tag{2}$$

The loglikelihood function of this model (apart from a constant) is given by

$$\ell(\beta, \sigma) = -\frac{N}{2} \ln \sigma^2 - \frac{1}{2\sigma^2} \sum_i (y_i - \beta' x_i)^2 \tag{3}$$

The maximum likelihood estimates of β and σ are the values $\hat{\beta}$ and $\hat{\sigma}$ that maximize function (3). In the following example we activate the maxlik-library, generate data according to model (2) and the maximum likelihood estimates of β and σ are determined using the `maxlik`-procedure:

```
library maxlik;
maxset;
```

[9] In this manual we deal with version 3.1.3 of the maxlik-library.

[10] A related third-party library is the cml-library to do constrained maximum likelihood estimation.

```
output file=maxlik1.out reset;
nobs=500;
beta=1|-1|0.5|-0.25;
s=2;
x=ones(nobs,1)~3*rndn(nobs,3);
y=x*beta+s*rndn(nobs,1);
theta0=ones(5,1);

/* set global variables */
_mlparnm="BETA1"|"BETA2"|"BETA3"|"BETA4"|"S";
_mlalgr=2;
_mlstep=1;
_mlcovp=3;
{x,f,g,cov,retcode}=maxlik(x~y,0,&loglik,theta0);
call maxprt(x,f,g,cov,retcode);

end;

proc loglik(theta,z);
 local y,x,b,s;
 x=z[.,1:cols(z)-1];
 y=z[.,cols(z)];
 b=theta[1:cols(x)];
 s=theta[cols(x)+1];
 retp( -0.5*ln(s^2)-0.5*(y-x*b)^2/s^2 );
endp;
```

Apart from a call to the `maxlik`-procedure, the user only has to program a function that has a column vector with contributions to the logelikelihood as output. The `maxlik`-procedure takes a pointer to this likelihoodfunction as one of its arguments and the function is optmimized (in fact, the negative of minus the loglikelihood is minimized) using numerical derivatives and default optimization algorithms.

The central procedure in the maxlik-library is the `maxlik`-procedure. This procedure has four arguments and its output consists of 5 elements:

```
{x,f,g,cov,retcode}=maxlik(dataset,vars,&loglik,theta0);
{x,f,g,cov,retcode}=maxlik(data,0,&loglik,theta0);
```

In the first line, the data are read from the file `dataset` and the vector `vars` is a character vector with the names of the variables used in the analysis or a numerical vector with the indices of the selected variables. In the second case, the data are passed as a matrix `data` (see also the example above). The third argument is a pointer to a procedure that returns a vector with contributions to the loglikelihood function. This procedure has two arguments: the first is the parameter vector and the second is the data matrix. If the data are read from file, the order of the columns in the data matrix is specified in the vector `vars`. The final argument of `maxlik`-procedure is `theta0`, a numerical vector with starting values for the optimization.

Note that in the example above the parameter vector of the `loglik`-procedure `theta` equals β and σ stacked on top of each other. Each row of the data matrix `z` consists of an observation $(\, x_i' \quad y_i \,)$ and correspondingly, each element of the output vector with contributions to the loglikelihood is equal to

$$-\frac{1}{2}\ln\sigma^2 - \frac{1}{2\sigma^2}(y_i - \beta' x_i)^2.$$

The first output vector, `x`, is the vector with estimated parameters. The scalar `f` is the value of the function at the minimum (the mean of minus the loglikelihood), the vector `g` is the value of the gradient evaluated at `x`, the matrix `cov` is the covariance matrix of the

parameters and the scalar `retcode` indicates whether the optmization has terminated normally (`retcode = 0`) or not. These results can be printed using the `maxprt`-procedure:

```
call maxprt(x,f,g,cov,retcode);
```

Optimization of the loglikelihoodfunction proceeds in two steps. At each iteration a direction d is determined along which line a 'better' value for $\hat{\theta}$ is searched, and a step length λ is calculated which improves the value of the likelihood function. The new value of $\hat{\theta}$ is then determined as $\hat{\theta}_{i+1} = \hat{\theta}_i + \lambda_i d_i$.

The direction is commonly computed as (omitting the subscript i indexing the iteration) $d = Ag$, with A some square matrix and g the gradient of the loglikelihood function. Different choices of A are available, they correspond to different optimization algorithms. The optimization algorithm is set by the global variable `_mlalgr`. A first choice of A is to set $A = I$, with I the identity matrix. This method is the steepest descent algorithm (`_mlalgr=1`). Another choice is to set A equal to the inverse of the Hessian of the loglikelihood function, which is known as the Newton-Raphson algorithm (`_mlalgr=5`). This is method is computationally rather computer intensive and works only well if the loglikelihood function is well-behaved. So-called secant methods approximate the Hessian of the loglikelihood function by adding updates to an Cholesky decomposition of the Hessian. This Cholesky decomposition is then used to determine a direction d. The approximation of the Hessian improves with the number of iterations. Three such secant methods are implemented: the Broyden, Fletcher, Goldfarb and Shanno algorithm (`_mlalgr=2`), a scale-free version of this algorithm (`_mlalgr=3`) and and the Davidon, Fletcher and Powel algorithm (`_mlalgr=4`). Another well-known optimization algorithm in econometrics is implemented: the Berndt, Hall, Hall and Hausman method. This method estimates the Hessian by the average of the outer products of the gradients of each contribution to the loglikelihood function. The default optimization algorithm as set in `maxset` is the BFGS-algorithm (`_mlalgr=2`) and for most problems there is no need to use another optimization algorithm as it is reasonably robust against scaling and conditioning of the model.

Given the direction d, one has to determine the step length. The function to be minimized by λ is $\ell(\theta + \lambda d)$. The way λ is calculated is set by the global variable `_mlstep`. The simplest method is to set $\lambda = 1$ (`_mlstep=1`). A better procedure is to fit a quadratic or cubic function (in λ) to $\ell(\cdot)$ and then to choose λ such that this approximating function is maximized (`_mlstep=2`). Other methods of obtaining a step length are based on a sequence of values of λ (`_mlstep=3, 4, 5`). The default choice is `_mlstep=2`.

Optimization of the loglikelihoodfunction stops if convergence is reached, or if the maximum number of iterations is reached. The latter is set by `_mlmiter` and the default maximum number of iterations is 10000. Convergence is reached when the tolerance of the gradient (which is defined as a constant times the largest element of the gradient vector) is below is below $1e-4$. The tolerance level can be set with the global `_mlgtol`. In case the optimization terminates with an error, the current values of the parameters is stored in the global variable `_mlcoef` and the Hessian is stored in the global variable `_mlfhess`. In case the Hessian fails to invert might want to check whether the model is identified by examining the eigenvalues and eigenvectors of the Hessian. Near the optimum the loglikelihood function can be approximated by

$$\ell(\theta) = \ell(\hat{\theta}) + (\theta - \hat{\theta})' H (\theta - \hat{\theta})$$

with H the Hessian. The Hessian can be decomposed as $H = \sum_j \psi_j t_j t_j'$ with ψ_j the eigenvalues and t_j the eigenvectors of the Hessian. Hence, the loglikelihood function near the optimum can be approximated by

$$\ell(\theta) = \ell(\hat{\theta}) + \sum_j \psi_j \left(t_j'(\theta - \hat{\theta}) \right)^2 .$$

If some eigenvalue ψ_k is very small then the linear combination $t_k'(\theta - \hat{\theta})$ can be changed 'a lot' without affecting the value of the optimum. Inspection of t_k might give insight whether

the flatness near the optimum is associated with one paramater of with a simple linear combination of some parameters.

The covariance matrix of the parameters can be estimated in three different ways. The first estimate is the inverse of the Hessian (_mlcovp=1), the second estimate is the inverse of the matrix of cross-products of first derivatives (_mlcovp=2) and the third estimate is White's (White (1982)) heteroscedasticity-consistent estimate of the covariance matrix (_mlcovp=3). The latter estimate can also be used for testing the specification of the model.

By default, the maxlik-procedure uses numerical derivates during the optimization. Of course, this may result in many evaluations of the loglikelihood function at each iteration. It is possible to supply a procedure that calculates the derivative of the loglikelihood function analytically. That procedure must have two arguments: a vector with parameters and a matrix with the data. The output of this analytical gradient procedure is a matrix with as many rows as the data matrix and the columns contain the derivatives with respect to the parameters. An analytical gradient is used by the maxlik-procedure if the global variable _mlgdprc is set to a pointer to that gradient procedure.

The derivatives of the loglikelihood function in the normal linear model are given by

$$\frac{\partial \ell_i}{\partial \beta} = \frac{1}{\sigma^2}(y_i - \beta' x_i)x_i$$
$$\frac{\partial \ell_i}{\partial \sigma} = -\frac{1}{\sigma} + \frac{1}{\sigma^3}(y_i - \beta' x_i)^2$$

where ℓ_i denotes the contributions of the ith observation to the loglikelihood. This analytical derivative is programmed as the procedure loglikgd below. Each row of the matrix returned fromt his procedure contains the derivatives given in the two equations above.

```
library maxlik;
maxset;

output file=maxlik2.out reset;

nobs=500;
beta=1|-1|0.5|-0.25;
s=2;
x=ones(nobs,1)~3*rndn(nobs,3);
y=x*beta+s*rndn(nobs,1);
theta0=ones(5,1);

_mlgdprc=&loglikgd;
/* test analytical gradient first */
{x0,f,g,cov,retcode}=maxgrd(x~y,0,&loglik,theta0);
/* now optimization using analytical gradient */
{x0,f,g,cov,retcode}=maxlik(x~y,0,&loglik,theta0);
call maxprt(x0,f,g,cov,retcode);

end;

proc loglik(theta,z);
 local y,x,b,s;
 x=z[.,1:cols(z)-1];
 y=z[.,cols(z)];
 b=theta[1:cols(x)];
 s=theta[cols(x)+1];
 retp( -0.5*ln(s^2)-0.5*(y-x*b)^2/s^2 );
endp;

proc loglikgd(theta,z);
```

```
 local y,x,b,s,e,gb,gs;
 x=z[.,1:cols(z)-1];
 y=z[.,cols(z)];
 b=theta[1:cols(x)];
 s=theta[cols(x)+1];
 e=y-x*b;
 gb=e.*x/s^2;
 gs=-1/s+e^2/s^3;
 retp( gb~gs );
endp;
```

Before the `maxlik`-procedure is called, the `maxgrd`-procedure is executed. This procedure has the same parameters as the `maxlik`-prcedure and it calculates the derivatives of the loglikelihoodfunction both using the user supplied procedure `loglikgd` and numerically. By comparing the output of the two, one can check whether the analytical derivative yields approximately the same results as the numerical derivative. If the reader ever programs an analytical derivative he is strongly advised to check his procedure using `maxgrd`. Of course, any maximum likelihood program can easily be checked with simulated data!

8 Graphics

GAUSS is equiped with some graphics capabilities. With a few commands one is able to plot two- and three dimensional data and the graphs can be saved for incorporation in a word processor. In the first subsection we will deal with the most important commands of the graphics library, in the second subsection we show how to include GAUSS graphics in a LaTeX document. It is possible to create multiple windows with each window containing a graph. This feature will not be discussed here. General help on the graphics library is obtained by asking help `@pqg` in the online help program.

8.1 Creating Graphics in GAUSS

In this section we discuss configuration of the graphics library and the most important commands. The graphics library of GAUSS is configured analogously to the way GAUSS is configured using the file `gaussi.cfg`. The configuration settings for the graphics library are stored in `pqgrun.cfg`. Some of the options set in this file can be changed while running GAUSS (all print options), but the video adapter type can not be changed at run time. GAUSS supports various printers: Epson, Citizen, HP LaserJet and DeskJet and Postscript. It is important that the directory that contains the file `gaussi.exe` is listed in the DOS search path. It will not be possible to run the graphics program if that is not the case.

The graphics library is activated by the `library pgraph;` command and global variables are set tot heir default values by the `graphset;` command. GAUSS graphics are created in four steps:

1. activation of the graphics library,
2. reading and formatting of the data,
3. setting global variables to special values,
4. calling the actual graphics procedure needed.

In this section we will focus on the third and fourth step. After a graph is displayed on screen the user has two options. Pressing `<ESC>` lets the program continue and pressing `<enter>` yields a menu with some saving and printing options.

`xy(x,y)`	`logx(x,y)`	`logy(x,y)`	`loglog(x,y)`
`bar(v,h)`	`hist(x,v)`	`histf(x,f)`	`histp(x,p)`
`box(g,h)`	`xyz(x,y,z)`	`surface(x,y,z)`	`contour(x,y,z)`

Table 2: Graph types

In table 2 we give the commands that generate graphics. The names of the procedures indicate the type of graph it creates.

Output is partly determined by global variables of the graphics library. Of course, these values must be set before the graph is drawn using one of the commands in tabel 2. Global variables are set by calling a procedure (for example, `title("example plot")`) or by explicit assignement (for example, `_pdate=0`). Some global variables have a similar effect on different graph types. Global variables are reset to their default values by calling the procedure `gaphset` (this is a procedure without arguments or returns). Axis can be named with `xlabel("text on x-axis")`, `ylabel("text on y-axis")`, and `zlabel("text on z-axis")` and a title is given with `title("title of the plot")`. The date and time of creation of the graph is printed in the upper left corner. This feature can be removed by setting the global variable `_pdate=0;`. If `_pdate` is set to a string, then this string will be printed in the upper left corner and the date will be appended.

The commands `hist(x,v)`, `histf(f,c)` and `histp(x,v)` all produce histograms. They differ only in the quantity displayed on the vertical axis (absolute numbers, frequencies of percentages) and input requirements (raw data and a vector of breakpoints (`hist` and `histp`) or frequencies and a vector with category labels (`histf`). In the example file below we generate a vector with normal distributed numbers and a histogram is generated using -1.5, -1, 0, 1 and 1.5 as breakpoints so that the first bar corresponds to observations $x \leq -1.5$, the second category to $-1.5 < x \leq 1$, etc.

```
library pgraph;
graphset;

x=rndn(100,1);
b=-1.5|-1|0|1|1.5;
title("histogram of normal distribution");
ylabel("frequency");
xlabel("x value");
_pdate=0;
call hist(x,b);
```

Two dimensional graphs can be plotted using the `xy(x,y)` command. Of course, the arguments `x` and `y` must be of equal length unless one wants to index the observations. The command `xy(1,y)` plots the curve $(1, y_1)$, $(2, y_2)$, etc. The second argument argument need not be a vector: if a matrix (with the same number of rows as `x`) is passed as an argument, each column will be graphed. Consider for example the following program

```
library pgraph;
graphset;

ngrid=100;
x=seqa(-3,6/ngrid,ngrid);
y1=pdfn(x);
y2=1/pi*1./(1+x.^2);

_plegctl=1~4~1~.32;
_plegstr="Gaussian density\000Cauchy density";
```

```
xlabel("ordinate");
ylabel("density");

call xy(x,y1~y2);
```

In this example, two density functions are graphed, viz. the density of the normal distribution and the density function of the Cauchy distribution. The second argument passed to the `xy`-procedure is a 100×2-matrix and each column is graphed as a seperate line. The legend in the graph is set by two variables. The 4-row vector `_plegctl` determines where the legend is placed. The first element determines whether the coordinates are set in plot coordinates (1), inches (2) or Tek pixels (3), the second element gives the font size of the legend (between 1 and 9) and the final two elements give the horizontal and vertical coordinate of the lower left corner of the legend (in units specified by the first argument). In our example, coordinates are given in plot coordinates, font size is 4 and the coordinate of the lower left corner is $(1, 0.32)$. The text of the legend is specified in `_plegstr`. Different curve labels are seperated by the ascii-0 character `\000`. The label of the first curve is 'Gaussian density' and the label of the second curve is 'Cauchy density'. If the number of labels given in `_plegstr` is less than the number of columns in `y` empty labels will be printed in the legend. Additional text messages can be plot in the graph using the global variables `_pmsgctl` and `_pmsgstr`. By its default settings (set using the `graphset` procedure, a line is drawn through the points plotted. The type of line plotted is controlled by the global variable `_plctrl`. If this variable is set to -1, only individual points are plotted and these points are not joined by a line. The symbol at each point is controlled using `_pstype`.

Three dimensional curves and contour lines are graphed using the `surface` and `contour` procedures. Both procedures have three arguments: a *row*-vector `x` of length k, a vector `y` of length p and a $p \times k$ matrix `z`. The rows of `z` correspond to the elements of `y` and the columns of `z` correspond to the elements of `z`. A graph of the surface above the $x - y$ plane is drawn by `surface(x,y,z)` and a two dimensional graph with isocontour curves is drawn by the command `contour(x,y,z)`. A three dimensional graph of a function is obtained with the `xyz(x,y,z)` procedure. This procedure yields 3D results analogous to the `xy` procedure discussed above. As an example we give the following program that generates three 3D plots based on a bivariate normal density function:

```
library pgraph;
graphset;

ngrid=51;
x=seqa(-3,6/ngrid,ngrid);
y=x;
x1=pdfn(x);
y1=pdfn(y);
z=x1.*y1';

xyz(x,y,z[.,1:2]);
surface(x',y,z);
contour(x',y,z);
```

Grpahics images can be saved to disk in one of the file formats supported using the `graphtprt` procedure. The argument of this procedure is a string that contains some flags. The most important flags are `-p` (print graph using the settings specified in `pqgrun.cfg`), `-pf=test` (print output to a file test), `-c=k` (convert file format to (`k`=1) Encapsulated Postscript, (`k`=2) Lotus PIC-file, (`k`=3) HPGL format or (`k`=4) PCX bitmap format) and `-cf=test` (print output to a file calles `test`). For example, the first graph drawn after the command `graphprt("-p -pf=test")` will be printed automatically to a disk file `test` using the printer default in `pqgrun.cfg`. Of course, the picture must be redrawn

before these settings take effect. This can be done explicitly by calling the graphing procedure of choice, but also using the `rerun` command. In the latter case, the last graph is redrawn.

8.2 Incorporating GAUSS Graphics in LaTeX Documents

Even though GAUSS is not capable of producing graphics that can be incorporated into LaTeX documents directly, it is not very difficult to use plots and figures in LaTeX documents. We assume that the user has a viewer and a printer program for LaTeX `.dvi`-files that are capable of displaying and printing `.pcx`-files.

The easiest way to incorporate GAUSS graphics in LaTeX documents is to save the graph in Encapsulated Postscript format (`.eps`) and to read it in the LaTeX document with the `epsfig`-package. The following LaTeX document is a simple example:

```
\documentclass{article}
\usepackage{epsfig}
\begin{document}
\begin{figure}
\epsfig{file=plotje,width=100mm}
\caption{Sample figure from GAUSS}
\end{figure}
\end{document}
```

A disadvantage of this approach is that a Postscript viewer and Postscript printer are necessary for viewing and printing.

Alternatively, one can incorporate GAUSS graphics by creating a bitmap first. It takes three steps to achieve incorporate a GAUSS graphic in a LaTeX text. First, the GAUSS picture must be saved on disk. Second, this file must be converted to a `.pcx`-file and third, the `.pcx`-file must be included into the LaTeX document.

First, the picture must be saved in a format that can be converted later. The most convenient format is the Hewlett Packard format. A plot can be saved to (for example) the file `pict1.hpp` using the GAUSS commands

```
graphprt("-c=3 -cf=pict1.hpp");
rerun;
```

The file `pict1.hpp` is a binary file in HPGL-format. This file can be converted to a `.pcx`-file using the program `hp2xx.exe`[11]. Suppose that the final document will be viewed and printed at a 600 dpi resolution, and that the width of the picture in the document will be 100mm, then the conversion command is:

```
hp2xx -mpcx -d600 -D600 -w100 -f pict2.pcx pict2.hpp
```

The number after the `-w` parameter is the width of the picture (in mm) in the document. The name after the `-f` parameter is the filename of the converted file. The program `hp2xx` creates both a `.pcx`-file as well an Encapsulated Postscript file. More information on this program can be obtained by giving the command `hp2xx`.

After the conversion, we have a `.pcx` that can be inserted into a LaTeX document (for example) by

```
\begin{figure}
\centerline{\putfigure{file=pict2,height=75mm,width=100mm}}
\end{figure}
```

[11] This program is distributed on the 4TeX cdrom, and can be obtained from the author.

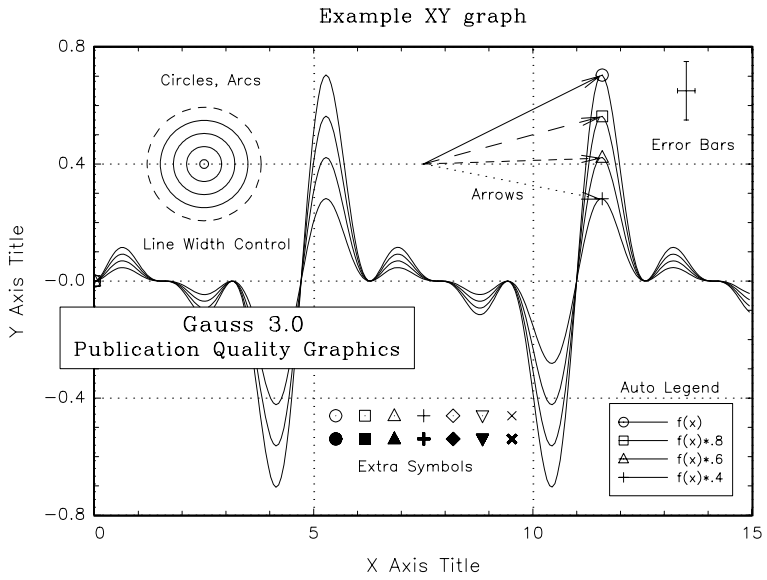

Figure 1: Gauss graphic from `examples/pxy.e`

The `\putfigure` macro is defined in the style file `figure.sty` and hence, that style file must be included. The height and width parameters in the `\putfigure` macro are mandatory, they 'tell' the TEX compiler the size of the figure. Of course, the width parameter should equal the `-w` parameter used by `hp2xx` to convert the picture to a `.pcx`-file.

In figure 1 we show an example of a GAUSS graphic incorporated into a LATEX text.

9 Exercises

1. Check your setup of GAUSS by running some of the example files in the `gauss\examples` directory. Have problems fixed by your system administrator.

2. Make a subdirectory for the programs and procedures you'll write during this course. Ensure that it is in the search path of GAUSS. Modify the file `startup` such that the working directory is set to this directory as soon as GAUSS is started.

3. Write a program that initializes a matrix with ones, a matrix U with uniform random numbers and one matrix with keyboard input. Count and print the number of elements of U that are smaller than 0.5.

4. Write a program that converts your day of birth (a string variable like `birth = "July-04-1966";` to a numerical vector with 3 elements, the first element representing the month, the second representing the day and the third representing the year.

5. Write a procedure that determines the length of a vector x.

6. Write a procedure that calculates the OLS estimator in the linear model, its covariance matrix and an estimate for the residual variance.

7. Write a procedure that has two arguments (a matrix and a string with a file name) and that save the matrix to an ASCII file with the specified file name.

8. In panel data econometrics one usually stacks the observations such that the individual index runs slow and the time index runs fast:

$$y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_T \end{pmatrix}.$$

Here, y_1 is an N-vector with the observations for all individuals in period 1, y_2 is an N-vector with all observations in the second period, etc. Write a GAUSS program that re-orders the vector y such that the time index runs fast and the individual index runs slow, *i.e.*

$$\tilde{y} = \begin{pmatrix} \tilde{y}_1 \\ \tilde{y}_2 \\ \vdots \\ \tilde{y}_N \end{pmatrix}.$$

Here, $\tilde{y}_1$ is a T-vector with all observations for individual 1, etc.

9. Write a procedure that calculates an N-vector with contributions to the loglikelihood in a Tobit model with N observations. Minimize the amount of computations required.

10. Write a program that generates a dataset according to the probit model:

$$\begin{aligned} y_i^* &= \beta' x_i + \varepsilon_i \qquad \varepsilon_i \sim \mathcal{N}(0,1) \\ y_i &= \begin{cases} 0 & y_i^* \leq 0 \\ 1 & y_i^* > 0 \end{cases} \end{aligned} \tag{4}$$

Only the x_i's and y_i are observed. Save the observations in a dataset.

11. Using this dataset, calculate desciptive statistics and estimate β using OLS.

12. Create a graph of the standard normal, t- and Cauchy-density functions. Save your graph and generate a print.

13. Write a procedure that calculates the contributions to the loglikelihood of a probit model. The first parameter of the procedure is the parameter β and the second part is a datamatrix Z.

14. Examine by means of a simple simulation study whether Glesjer's test for the presence of heteroscedasticity in a linear regression model is more powerful than an omnibus test like the Breusch-Pagan test (see Greene (1993), p. 394–396).

References

Dennis, J.E. and R.B. Schnabel (1983). *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Englewood Cliffs, N.J.: Prentice-Hall.

Greene, W.H. (1993). *Econometric Analysis* (Second ed.). New York: MacMillan.

Press, W.H., S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery (1992). *Numerical Recipes in C* (Second ed.). Cambridge: Cambridge University Press.

White, H. (1982). Maximum likelihood estimation of misspecified models. *Econometrica 50*, 1–25.

A Converting ASCII to GAUSS datasets

An ascii dataset has to be converted to a GAUSS dataset before it can be used efficiently. To facilitate this conversion, the utility `atog386.exe` has been provided. This utility can be operated both interactively and in batch mode. In case one uses the utility interactively, it is started by `atog386` at the command prompt. The utility returns with a prompt \ after which a limited number of commands can be entered (see below). All commands are to be terminated with a semicolon (`;`). An interactive session is ended by the command `quit`.

In batch mode, the conversion to a GAUSS dataset is performed by a command like `atog386 convert.cmd`. The file `convert.cmd` contains the commands that perform the conversion. The most useful commands[12] are

append; append ascii dataset to an existing GAUSS dataset,

help; display help (interactive mode only),

input parameters name of the ascii dataset,

output parameters name of the GAUSS dataset,

msym parameter missing value character (default dot '.'),

invar parameters specification of data in ascii dataset,

outvar parameters list of names of variable to be saved in GAUSS dataset,

run; execute commands entered (interactive mode only).

The most crucial command is the `invar`-command. This commands tells `atog386` how the data are organized in the ascii dataset. ASCII datasets may either be softly delimited (elements are separated from each other by spaces, tabs or carriage/returns-line feeds), hard delimited (elements are delimited by a printable character) or not be delimited at all. In the first case, the command reads as

```
invar $name #age weight $char[1:10] #numvar[03];
```

The first variable is a character variable (indicated by the $-sign), the second and third variable are numerical variables (indicated by the #-sign), the next ten variables are labelled `char1` to `char10`, and finally, the last three variables are numerical again and labeled as `numvar01`, `numvar02` and `numvar03`.

If the ascii dataset is hard delimited, the format changes to something like

```
input data.csv;
output datacsv;
invar delimit(,,N) #var1,var2,var3,var4,var5;
outvar var1,var2,var3,var4,var5;
```

The keyword `delimit` has two optional parameters: the first is the delimiter (`!` in this example,use `\r` if each line contains one record and the variables are seperated by commas) and the second one indicates whether the last element of an observation is followed by a delimiter (in which case it takes the value `N`). The program assumes that the last variable is not followed by a delimiter if no second argument is specified. Note that the values taken character variables may not include spaces. `atog386` reports an error if it encounters an input line like

```
34,s korea,belgium,0,0,32
```

[12] For a detailed description we refer to the GAUSS manual.

because of the space in the second field. If the data are organized as packed data with fixed records length there are no problems if a character variable takes a value with a space in it.

A packed ascii file must have records with fixed lengths. The keyword `record` is used to indicate the length of the record (including the final CR/LF, which takes two positions). Furthermore, the format of the variables must be specified:

```
invar record=47 $(1,4)name #(5,2)age (7,4.2)weight $(*,1)char[1:10]
 #(*,8.2)numvar[03];
```

The first variable (`name`) starts at position 1 and has length 4. The `age` variable starts at position 5 and has length 2. The starting position of the `weight` variable is 7, it occupies 4 bytes, of which the last two are decimals. A decimal point is inserted between the second and third element. The next ten variables are all character variables of length 1. The asterisk in the format field indicates that the succeeding field starts immediately after the preceding field. The last three variables are numerical, have length 8 and the last two numbers are converted to decimals. A record has 45 fields and is terminated by a `<CR><LF>`, so the total length is 47 bytes.

An example of a complete command file for `atog386` is

```
input c:\gauss\data\wbo.asc;
output c:\gauss\data\wbo;
invar huur inkomen ihs respnr;
outvar huur inkomen ihs respnr;
msym &;             @ missing symbol@
```

Note that no double backslashes are used in naming both the input and the output file, the variables `invar` and `outvar` are literals. Comments are enclosed in @-signs. The command `atog386 convert.cmd` converts the ascii file `wbo.asc` to the GAUSS dataset `wbo.dat` and accompanying header file `wbo.dht`.